Concepts and Tools for the Software Life Cycle

R. C. Tausworthe
DSN Data Systems

The Tools, techniques, and aids needed to engineer, manage, and administrate a large software-intensive task are themselves parts of a large software base, and are incurred only at great expense. This article focuses on the needs of the software life cycle in terms of such supporting tools and methodologies. The concept of a distributed network for engineering, management, and administrative functions is outlined, and the key characteristics of localized subnets in high-communications-traffic areas of software activity are discussed. A formal, deliberate, structured, systems-engineered approach for the construction of a uniform, coordinated tool set is proposed as a means to reduce development and maintenance costs, foster adaptability, enhance reliability, and promote standardization.

I. Introduction and Background

In 1979 it was recognized that more than half of the United States' National Aeronautics and Space Administration (NASA) budget was required to sustain labor-intensive, routine services and operations, and that fraction was rising. In 1980, the NASA study committee on machine intelligence and robotics identified the potential of computer science for increasing the productivity and the affordability of future space operations. Computer science was identified as the most critical supporting technology requiring an intensive NASA research and development involvement. The committee recommended that NASA develop a long-term commitment and a centralized, coordinated effort to alleviate the budget drain, support a higher-caliber personnel skill mix, and enable productivity breakthroughs that would enable expanded mission sophistication.

The world's entire industry is rapidly becoming information-intensive, silicon-based, and software-driven[1]. The trends in modern information systems are toward very large data bases, distributed systems, computer networks, less expensive hardware, complex applications, and automated workplaces and factories. The space effort administered by NASA, in many respects, is but a particular example of this tendency. Although its progress may not, perhaps, be as pronounced as some research organizations within the USA and elsewhere abroad (Ref. 1) have reported, NASA, nevertheless, has recognized the need to capitalize on computer technology advances to increase the affordability of its coming missions.

[1] Reifer, Donald, session-opening comment at COMPSAC-79, Chicago, Ill., November 1979.

People-costs are rising at inflationary rates – 5% to 15% per year – while computer logic costs are decreasing some 25% per year and computer memory costs are decreasing about 40% per year. Overcoming the labor-intensive nature of building and maintaining large, complex information systems increasingly rests on raising the productivity of personnel responsible for the *software* (i.e., programs, data bases, and documentation) within the system. Software efforts are not yet mechanized to the point that their products are "manufactured" in production-line fashion.

A. Software Problems

Software, from its beginnings, has been immersed in the same kinds of problems any new kind of industry undergoes. Software products have too often been late and over-budget, often have not fulfilled the needs of users, often have been unreliable, often are difficult to modify to meet new user needs, and often cannot be adapted to accommodate evolving technology. The useful lifetime of that software is therefore often shorter than planned, usually at low salvage value upon its retirement.

Even though many "software problems" are, in reality, perhaps better classified as "systems engineering problems," the fact is that systems are becoming so complex that the elaboration and management of system details are becoming literally impossible without computational aid. That is, system engineering relies on being able to find solutions to system problems via software. Hence, software problems permeate the entire application system in which they reside and the entire development system in which they are created.

If software problems could be localized, or related to simple, noninteracting influences, or could be found to be coupled to simple, fundamentally wrong underlying principles, perhaps they could be attacked with more sweeping results. But the large software project operates within an entangled socio-economic system that must respond to intangible or subjective requirements, accommodate an uncertain discovery/development process, communicate excruciatingly detailed information through an imprecise, ambiguous human medium, and perform within sometimes severe fiscal and schedule constraints to build or maintain a very large, technically complex, intricate application system. Consequently, there is almost a complete fabric of austere, tightly interwoven technical and managerial problems entangling each programming project in the large.

Because software costs are rising, this rise being primarily related to labor, software costs often dominate the system costs. For a number of reasons, cost and schedule performances in software tasks have been regarded as poor in comparison with hardware tasks. Software tasks are also generally acknowledged to be harder to manage than hardware tasks. Furthermore, qualified software implementation and management personnel are scarce.

Labor-intensive efforts are reduced by improvement in methodology, environments, tools, and aids – what we shall refer to in the current context as *software engineering technology*. Software engineering is a disciplined approach to software-intensive efforts whose goal is to solve software and related technological and administrative problems in an organized, responsible, professional way. Software engineering also seeks to improve the quality, performance, and usability of computational facilities, as well.

A recent study (Ref. 2) by the University of Maryland reported the results of a survey of state-of-the-art technology and its utilization in the United States and Japan. Its principal conclusion was that, while researchers in these two countries took advantage of, and further expanded the state of the art, the large production programming projects within the same organizations did not. Cited as reasons for not using up-to-date tools and methodology were unawareness and lack of training, poor human engineering of the tools, lack of good documentation and support, unreliability, and cost overheads.

As a result, the conclusions of the NASA study committee still seem to describe most of the software industry today. True, progress may have been made in the interim, but it appears outwardly that resultant productivity has been little affected, perhaps counteracted by rising applications and systems complexity. There still seems to be about a 10-year lag between state-of-the-art software engineering technology and its propagation into accepted practice. And software organizations need an injection of new technology in excess of 10% per year just to remain 10 years behind!

The software situation may be becoming increasingly more serious. It was described in the 1960s and early 1970s as "the software problem," but the term escalated to "the software crisis" in the late 1970s and early 1980s. The "continuing software crisis" is now expected to be with us throughout the 1990s (Ref. 3).

B. Critical Near-Term Needs

Some of the critical needs within the state of the art are: establishment of broader funding bases of support within organizations for the acquisition and integration of software engineering technology; improvement in the usage, quality, and effectiveness of software tools; betterment of the software implementation and management processes; provisions for better education and training in the use of software technology; and adoption of standard practices in software efforts.

Training of software personnel in the application of modern software engineering methods and standard practices throughout the entire software life cycle is certainly needed, but very costly to perform on the job. Implementing software tools for technology transfer by giving proper attention to portability, human engineering, adaptability, and product standards requires particular emphasis, but again requires objectivity, training, and monitored adherence. Improved hardware facilities, support software and operating systems, and networked work stations are incurred also at extra cost and complexity.

Means of coping with problems that are not inherently software-caused may also be relieved to some degree by increased understanding and automated support. Problems attributable to uncertain, ambiguous, and unstable user or system requirements may cause less frustration if risks could be quantitatively assessed and compensations made in costs, schedule, and program design. Some software engineering methods and tools, such as those for software requirements analysis and structural design, may be extensible, to some degree, to systems design and systems engineering tasks, even those not involving imbedded software.

C. Goals and Directions

Increased manpower effectiveness is essential to lowering computing costs and mission costs over the remainder of this century. To maintain affordability amid the increasing information system complexity that will accompany its coming space missions, NASA is considering the goal of a 500% increase in manpower effectiveness over the next 20 years.

Manpower effectiveness is increased primarily by methods and tools. Because software costs are not concentrated in any one particular phase of activity, it will be necessary to provide such means throughout the entire software life cycle. Tools concentrated in a particular phase can have only limited benefit. For example, the existence and use of a miraculous language that could render the coding activity instantaneous by itself would yield a cost benefit of only about 20%, because only about 20% of the current software effort is coding.

The 1980 NASA study committee cited the following development needs: the expansion of on-line, interactive, and automatic programming to increase productivity; a modern data-structuring language; symbolic models and computer representations of application areas of knowledge-based usage systems for these application areas; and the use of an integrated design, programming, documentation, and management data base. They concluded that the systematic, thorough application of automation technology to NASA was essential to the agency's cost effectiveness; that current technology could significantly reduce staff and response time; and that the use of computer-based machine intelligence could further increase productivity and utility of acquired information.

A number of efforts in private industry and government, both in the United States and abroad, have for some time now focused growing concern on software problems. There are three current concerted efforts led by the Department of Defense, and conjoined by NASA and certain constituents of private industry, to provide a common programming language (Ada[2], Ref. 4), to provide Software Technology for Adaptable, Reliable Systems (STARS) (Ref. 5), and to integrate Ada and other software technology into practice (the Software Engineering Institute). These efforts are oriented not only toward developing better software engineering technology, but also toward accelerating its propagation into the software industry.

A current study sponsored by the U.S. Air Force (Ref. 3) is evaluating the needs, issues, costs, benefits, and risks of a comprehensive standardized software engineering life-cycle support environment for the STARS program. Such an environment, when implemented, states the report, should have a dramatic positive impact on the continuing software crisis. However, the costs of the STARS program are large, about $300 million.

Unfortunately, a single standard environment and set of support tools may not work for everyone. The computing field may be just too diverse for that. The costs of a custom, complete programming environment for each application area are probably prohibitive for all except the large software houses. The development of a modern environment of the STARS variety is beyond the capital resources of even the large software houses.

Others must therefore content themselves with commercial, STARS-fallout, public-domain, or government-supplied environments, plus the smaller special tools that they must have and can develop for themselves. It therefore behooves organizations to adopt standards that will promote the infusion of this technology at the least cost. An example in which this approach was successfully applied has been recently reported by TRW (Ref. 6) in its Software Productivity System.

II. Software Technology Improvement Areas

As new software engineering tools or techniques emerge, they may or may not become a part of a particular software engineering environment. If they are not integrated into an

[2]Ada is a trademark of the U.S. Government (Ada Joint Program Office).

environment and properly used, they are ineffective, regardless of their potential. Improvements to the general software situation will therefore come about primarily through improved software engineering environments. Let us, for a moment, ponder some possible candidates for this environment and certain of their characteristics.

A. Near-Term Improvement Areas

Since most software organizations are not currently benefiting from existing technology anywhere near the maximum extent possible, probably the least expensive alternative for productivity and quality improvement is making off-the-shelf tools and techniques usable. This alternative requires awareness and existence of a spectrum of compatible, proven tools and techniques, training in their usage, perhaps better human-machine interfaces, and good vendor support.

In addition, there may be other worthwhile concepts within the current state of technology that can be readily implemented, given funding and the same engineering care as referred to above. Regardless of the alternatives, timeliness, benefits, and costs will determine the members of the tool set acquired.

Tools in this term may be likened to ordinary "hand tools," in that they are primarily automated instances of the routine kinds of things that people do. Their mechanical advantage is significant, but not outstanding.

Items in this category include programmer's toolkits, workbenches, and operating systems; management planning and status-monitoring tools; prototypes of requirements and design languages and analyzers; code and documentation auditors; software life-cycle mathematical models; design and documentation data bases; interactive environments for programming and management; test-case generators and test-path monitors; assertion validation monitors; reliability assessment models; regression data bases; and software engineering standard practices.

B. Medium-Term Improvement Areas

In a somewhat longer time frame, more ambitious and more costly "power tools" of the "computer-assisted, intuition-guided" variety may become generally available. The technical problems for such tools are definable today, but the solutions may require some technology development. Most tools in this class will probably be characterized by knowledge of the application built into (or available to) each tool. In such cases, this knowledge-assist is expected to yield a good mechanical advantage.

Tools in this category include such things as context-knowledgeable programming and document generation tools; software application data bases and reuse tools; sophisticated, accurate software life-cycle process phenomenology models; coordinated software production and task management tools; automated configuration management tools; program design quality analyzers; test and validation tools; requirement-code-test-configuration traceability tools; and friendly requirements capture-and-analysis tools. Many of these are in the process of being perfected today.

C. Long-Term Improvement Areas

The application of artificial intelligence techniques, specifically knowledge-based expert systems, to increase the automation of specification, design, implementation, and testing and validation processes probably yields the best hope of improving productivity and software reliability by at least an order of magnitude. But such tools will be very expensive to develop.

A knowledge-based system for software production would contain sets of software engineering rules integrated into an environment that would permit application of the rules to embryo software systems through the full range of life-cycle disciplines, technical and managerials. Tools of this type would play the role of an automated assistant in software development and maintenance, and would provide such services as conversational requirements capture and analysis; expert design consulting; expert software management decision support; self-coding documentation; automatic validation, failure analysis, and work-arounds; and direct requirements-to-product synthesis.

D. Prognosis

Use of the computer as a means to solve the problems caused by its own existence (as well as those thrust upon it by system engineering deficiencies) will necessarily be evolutionary, because the developed technology, being software, will only add to the problems it is aimed at curing. But it is believed that, within the STARS program, the technology-bootstrapping process will eventually converge to provide a considerable improvement in software productivity in general, and perhaps an order of magnitude in some areas, by the end of the century.

Current and near-term software engineering technology, if developed and *used*, can do much toward improving the production and management of software applications. Medium-term technology will potentially relieve programmers and managers of most burdensome activities that can be automated. If long-term technology is successful in removing currently known machine-intelligence limitations, programmers and managers, as we conceive of them today, may be put out of work; such individuals might be needed only for consulting and very special applications during the acquisition of requirements.

The effort in requirements capture seems to be a fundamental limitation, linked to the human discovery of need, uncertainty of the true form of the satisfaction needed, and the decision to act. Someday, perhaps someone will discover that the human channel capacity for requirements definition can be quantified by some Shannon-type (Ref. 7) information-theoretic limit. Then, just as the communications engineer now knows how to design communication systems optimally within this capacity, the software engineer will be able to fabricate systems to compensate for disruptive requirements, and software managers will plan for this uncertainty as a matter of course.

Of course, it is also possible that software engineering as a discipline, if not wisely administered, will die a horrible, lingering death – poisoned by wasted and fragmented efforts, crushed by an evolving enormity too great to be compatible within itself, strangled by overcontrolling managers, or starved by sponsors unwilling to invest capital to cut labor. Steps must be taken and attitudes must be adjusted to see that these kinds of things do not happen. Technological improvements must be planned, developed, capitalized, nurtured, and integrated into practice when they are ready. They will not come about by themselves; it will take continuing effort to make large software application systems affordable.

III. Software Production Information System

At this point, let us focus on a particular software environment, that in which the DSN Data Systems are implemented. The "Mark IVa" configuration for 1985 (Fig. 1), now in implementation, will contain some 145 computers of various kinds, and perhaps about 1.3 million lines of source code. The software resides in interconnected subsystems for deep-space tracking and data acquisition, spacecraft command, network performance validation, and communication system control and data routing.

The software production activity to support this configuration appears, at its upper abstraction, as a highly orchestrated, information-intensive process; the participants and their informational needs are shown in Fig. 2. From this viewpoint, the DSN environment perhaps looks fairly normal, much the same as any other environment.

However, within the environmental "black box," one finds a more complex, high-communications-traffic beehive of activity. For many reasons, partly inheritance, partly evolution, and partly because of operational constraints, software is developed on a number of different hosts, with the documentation and source code data base spread out over many types of media in different formats and on different machines. The flow of information and products in the implementation process is depicted in Fig. 3. This awkward, disconnected kind of environment, too, may look familiar to many. The reader will note that, while certain aspects of the development are automated, the communications among the various parts represent a serious deterrent to productivity.

The DSN therefore is developing a rationale and an architectural concept to restructure the software engineering environment for improved software throughput. That rationale and architecture are the subjects of the remainder of this article.

The software engineering environment is envisioned as an information system, an integrated set of processing and data federated around common interface considerations into an overall design that is balanced to serve the informational needs of all of its users. However, there are some constraints that prevent a drastic change from the current host mainframes and operating systems being used for software development and task management into a more modern development environment rife with tools, such as the UNIX[3] system. Changes must be made gradually (budgetary and training restrictions) and within the environment as it is being used for Mark IVa development.

The users of this information system for a given project include programmatic and institutional management, supervisors, system engineers, software engineers, quality assurance, administrative, and clerical personnel, user organization representatives, operations personnel, and various support staff, as well as those charged with implementing and maintaining the environment itself. Each has computational and communicational needs served by the environment.

Figure 4 shows the potential simplification in traffic obtained by providing users with workstations and other computational elements interconnected by local area networks, and providing object-oriented "servers" for major resources. Figure 5 shows how the management and administrative data bases can be organized around generic life-cycle functions and Fig. 6 shows how the same kind of organization can be applied to the software products and technical information. We have termed this configuration the Management and Development Network (MADNET). Prototype demonstrations of MADNET concepts were performed by Fouser[4].

The software engineering environment is envisioned as a layered architecture in which the user interface, at the top, is

[3]UNIX is a trademark of American Telephone and Telegraph, Inc.

[4]Fouser, T. J., Management and Development Local Area Network Concept Report, Report D-857, Jet Propulsion Laboratory, Pasadena, Calif., April 1983 (JPL internal document).

insulated from hardware, system software, and communications vagaries, and, at the bottom, by levels of virtual machines (Fig. 7). The user interacts with the system seemingly at the tool level using standard tool interfaces, although the tools themselves may be distributed within the system. Existing and purchased-off-the-shelf tools can be made more effective by imbedding them thus in a richer environment, by increasing their availability and accessibility, and by insulating the users from particular details of the host operating system(s).

A spectrum of tools is required to cover the range of users throughout the life cycle. Whereas the long-term goal is to build a fully integrated environment, short-term needs and limited resources make it necessary to provide first for integration of the tools available and readily acquirable. Thus, tools have been ported and purchased that are not 100% compatible with each other, but still serve users fairly well. Many of the Kernighan and Plauger Software Tools (Ref. 8) have been installed, and are in use.

Table 1 lists the kinds of tools that are needed and available (or could be made available in the near term) to development, management, and administrative personnel through the life-cycle phases. It also shows, upon analysis, where tools are absent and better tools are needed. (The Kernighan and Plauger Software Tools are not shown). Current tools tend to be clustered around the production phase for use by developers.

IV. Environment for Tool Development

The environment for building software tools need not be the same as the environment used to run the tools, nor is either of these necessarily the application software environment. We may thus make a distinction among software engineering environment, applications environment, and tools engineering environment. Each may be optimized toward its own ends.

The applications software, when operational, is mostly imbedded within a real-time data-acquisition system, and its lifetime is coupled to the lifetime of the space mission and the surrounding deep-space station hardware. Only in the event of a major redesign of the network would the transport and reuse of software in new computers be of concern.

However, a software tool that is generally useful over a number of projects needs to present the same user interface, regardless of its host or the application target machine. Toolware thus needs to be transportable to, or available on (or through), any workstation. In the interests of reducing retraining costs, learning to use a new tool, e.g., a new text editor or a radically different language dialect, just to work with a different host machine, should not be required.

A software layering technique can be applied to the construction of toolware, as indicated in Fig. 8. The figure shows a set of user tools built on a machine-independent set of functions written in an efficient machine-independent subset of a high-level language. These functions and language subset form a virtual machine, the *tool-builder's interface.* Below this interface, there will be a layer of functions with some degree of system dependency, which can be customized by suitable parameterization and minor modifications to present a stable *virtual machine layer interface.* At the next layer are the core toolset language and core system interface library of functions that form a *system and network layer interface.* This layer interfaces directly with the operating system, stabilizing many of the system dependencies into mere idiosyncracies that can be necessary to augment the existing operating system with "primitives," or host-dependent, application-independent code at the system-dependency level (Ref. 8).

Current investigations within the DSN are considering the use of the C language and the UNIX operating system (or UNIX-look-alikes) for building the DSN tools and tool interfaces. (Ada is not yet available on any of the machines in the current environment.) Non-UNIX-like operating systems tend to present problems only in certain file-access routines, which can be softened, to some extent, by the addition of auxiliary functions to create UNIX-like directories. Functions at the portable and customizable layers are maintained in source form and installed on each host to form a common library, so as to ensure the same operation of tools. Selectable options within the source media permit tools to capitalize on features of terminals, printers, and file systems, and yet not destroy the commonality of the user interface.

V. Tool Design Goals and Criteria

User-interface compatibility and, therefore, transportability are mentioned above as driving considerations in tool design. But a tool must primarily be *effective.* If making a tool portable also renders it ineffective, nothing is gained. Trade-offs among design goals must be made when conflicts occur.

There is a high priority for tools giving a significant and *measurable benefit* to the software engineering process. Since various factors tend to demotivate those in charge of funding the improvements to the software process, the benefits must be clear and capable of being demonstrated, and the expenditure must be justifiable.

In addition, tools should *interface well with other tools* and with the implementation methodology being applied within the software organization. When tools are built in-house or under contract, their interfaces can be prespecified. However, acquisiton of off-the-shelf items do not afford this opportunity, but it may be possible to adapt the tool via special install-options, or patches, or by way of a special version from the vendor, to the interface needs of the environment. In some cases, a special separate server may be developed to provide the input/output or function interface required. Figure 9 shows a layering of computational elements arrayed for server equalization of user interfaces.

Each tool should provide *wide-spectrum benefit* to the developer, to task management, to configuration management, and to quality assurance. For example, a good program design language tool can be used by the programmer to develop the structure, data flow, and detail design of program parts; it also serves to document this design; the manager may use statistics gathered by the tool as a status base for controlling the task; change detection algorithms in the analyzer assist configuration management; measurements of design complexity and automatic checks for completeness, for traceability to requirements, and for conformance to standards aid in quality assurance.

Tools should provide *status and quality information* relative to the object being worked on. This status should be unobtrusively extracted, as an integral part of the tool design (Fig. 10), and should automatically be made available to the management tools by way of a status data base.

Tools should be capable of being operated *interactively* through an appropriate friendly interface where intimate contact with a particular tool is required, such as when the details of an object are being worked out. There should also be a *non-interactive mode* to suppress all the details, once worked out. For example, a tool that has a set of interactive options for operation may skip the option-selection step the next time it is executed with the same object. Another approach is that of scripts executed by the operating system, as by the UNIX shell.

Tools should *have good life-cycle support*, and be as well engineered and documented as the products they support, or perhaps better. They should be built with usability and quality as goals. User manuals should be particularly well written and operation reliable. Moreover, since the tool (and documentation) will no doubt adapt to new applications, new methodologies, and new interfaces with other tools, it is important that the tool software be designed and documented for maintainability.

To the extent possible, *intermediate results and routine decisions should be hidden* from the user; final results should be immediate. For example, the edit-compile-link-execute-observe-re-edit cycle could conceivably (with sufficient computing power) be all integrated together merely into edit and observe windows, with all the intermediate compiling and linking steps suppressed, and with observed results almost simultaneously displayed.

VI. Summary

As information systems become more sophisticated and complex, the very means to make them affordable becomes an enabling technology. The tools that serve these means will themselves generally be sophisticated, complex, and costly. Thus, tool costs will generally have to be amortized across many projects tools to justify their capital expenditure.

Software tools then, perhaps more urgently than applications programs, require focused attention and concentrated effort to make them rehostable among many environments or to make them generally available within a distributed environment. The payoffs, however, can be significant in utility, training costs, tool acquisition costs to individual projects, schedules, and product quality. The layered-interface object-oriented approach in tool construction is one way to promote this rehosting, and standard network layering of protocols can help make the tools available throughout a distributed implementation environment.

References

1. Kim, K. H., "A Look at Japan's Development of Software Engineering Technology," *Computer*, Vol. 16, No. 5, May 1983, pp. 26-37.
2. Zelkowitz, Marvin B., et al., "Software Engineering Practices in the US and Japan," *IEEE Computer Magazine*, Vol. 17, No. 6, June, 1984, pp. 57-66.
3. Vick, Charles R., et al., *Methods for Improving Software Quality and Life Cycle Cost*, Report of the Committee on Methods for Improving Software Quality and Life Cycle Cost, Air Force Studies Board Commission on Engineering and Technical Systems, Washington, D.C., May 18, 1984 (draft).
4. Barnes, J. G. P., *Programming in Ada*, Addison-Wesley Publishers, Ltd., London, 1982.
5. Druffel, Larry E., et al., "The DoD STARS Program," *IEEE Computer Magazine*, Vol. 16, No. 11, November, 1983, entire issue.
6. Boehm, Barry W., et al., "A Software Development Environment for Improving Productivity," *IEEE Computer Magazine*, Vol. 17, No. 6, June, 1984, pp. 30-42.
7. Shannon, Claude E., "Communications in the presence of noise," *Proc. IRE*, Vol. 37, No. 1, January 1949, pp. 10-21.
8. Kernighan, Brian W., and Plauger, P. J., *Software Tools in Pascal*, Addison-Wesley, Reading, Mass., 1980.

Table 1. Support for software implementation

Personnel level	Implementation phase					
	Software planning and requirements	Software design definition	Software design and production	Section combined subsystem test	Acceptance test and transfer	Operation and maintenance
Administrative	WAD, SRM, MAIL Action items Procurement Travel, calendar	MAIL, SRM Action items Procurements Travel, calendar	MAIL, SRM Action items Travel, calendar	MAIL, SRM Action items Travel, calendar	MAIL, SRM Action items Travel, calendar	MAIL, SRM Action items Procurements Calendar
Management	WAD, SRM, MAIL Action items Requirements capture Requirements analysis ECR/ECO Procurements Review preparation aids Softcost	WBS, MAIL Action items Software visibility Productivity metrics Traceability metrics Design quality metrics Review preparation aids	WBS, MAIL, ARS, DVCS Action items Software visibility Productivity metrics Traceability metrics Design quality metrics Review preparation aids	WBS, MAIL, ARS, DVCS Action items Software visibility Productivity metrics Traceability metrics QA metrics	WBS, MAIL, ARS, DVCS Action items Software visibility Productivity metrics Traceability metrics QA metrics	ARS, MAIL, Action items ECR/ECO Transfer status DB
Development	Word processing Requirements capture Requirements analysis ECR/ECO SPMC documentation/ graphics Graphics	Word processing Requirements analysis PDL, CRISP ECR/ECO SPMC documentation/ graphics	EDIT, POL, CRISP, PASCAL, HAL/S, PL/M, MODCOMP assembly, DVCS SPMC documentation/ graphics Simulated test environment Debuggers	EDIT, PASCAL, HAL/S, PL/M MODCOMP assembly DVCS, STAR test generator SPMC documentation/ graphics Simulated test environment	EDIT, PASCAL, HAL/S, PL/M, MODCOMP assembly DVCS, STAR test generator SPMC documentation/ graphics Regression tests	ARS ECR/ECO

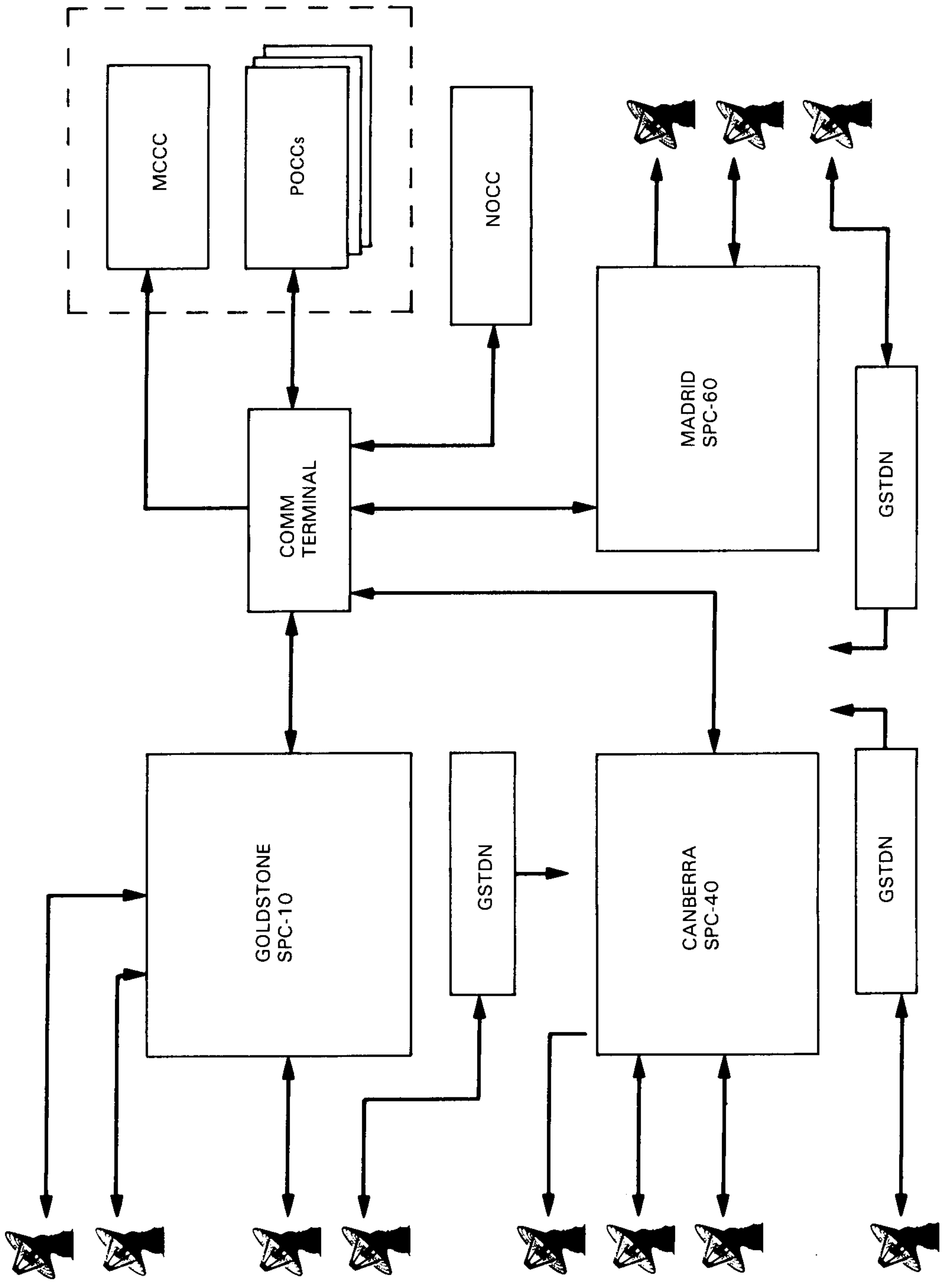

Fig. 1. Deep Space Network, Mark IVa, 1985

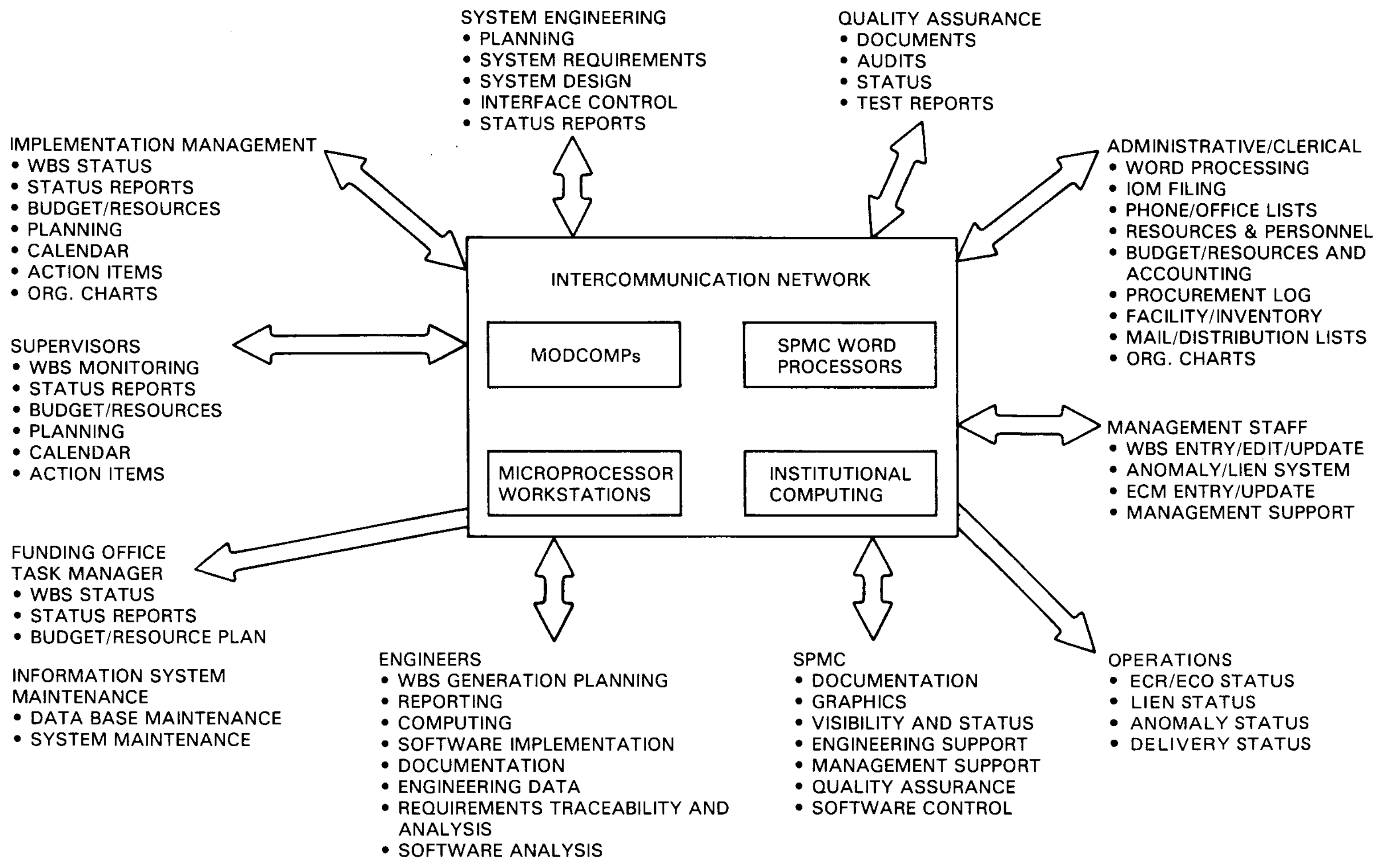

Fig. 2. DSN production information system

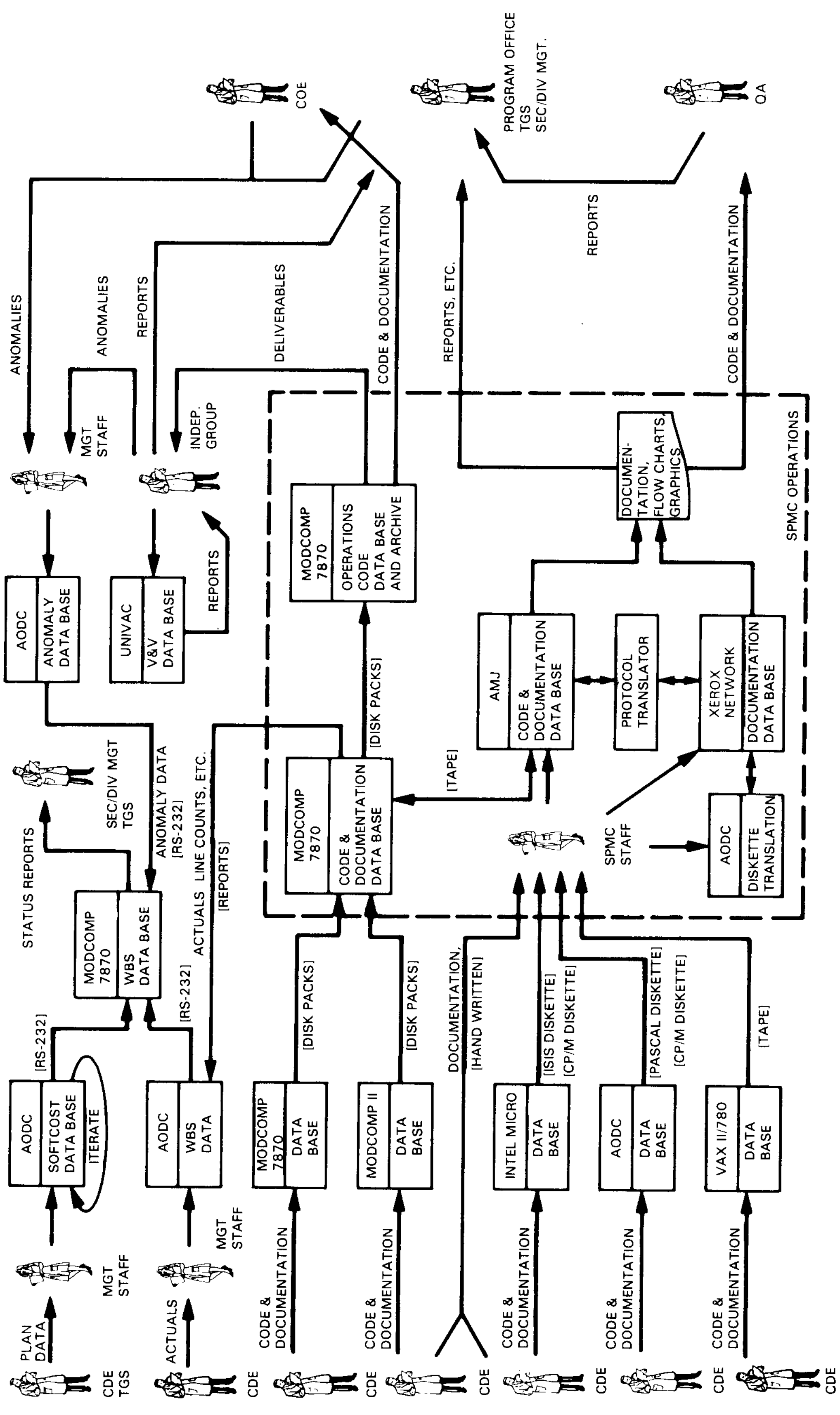

Fig. 3. Early DSN Mark IVa implementation process flow (from footnote 4)

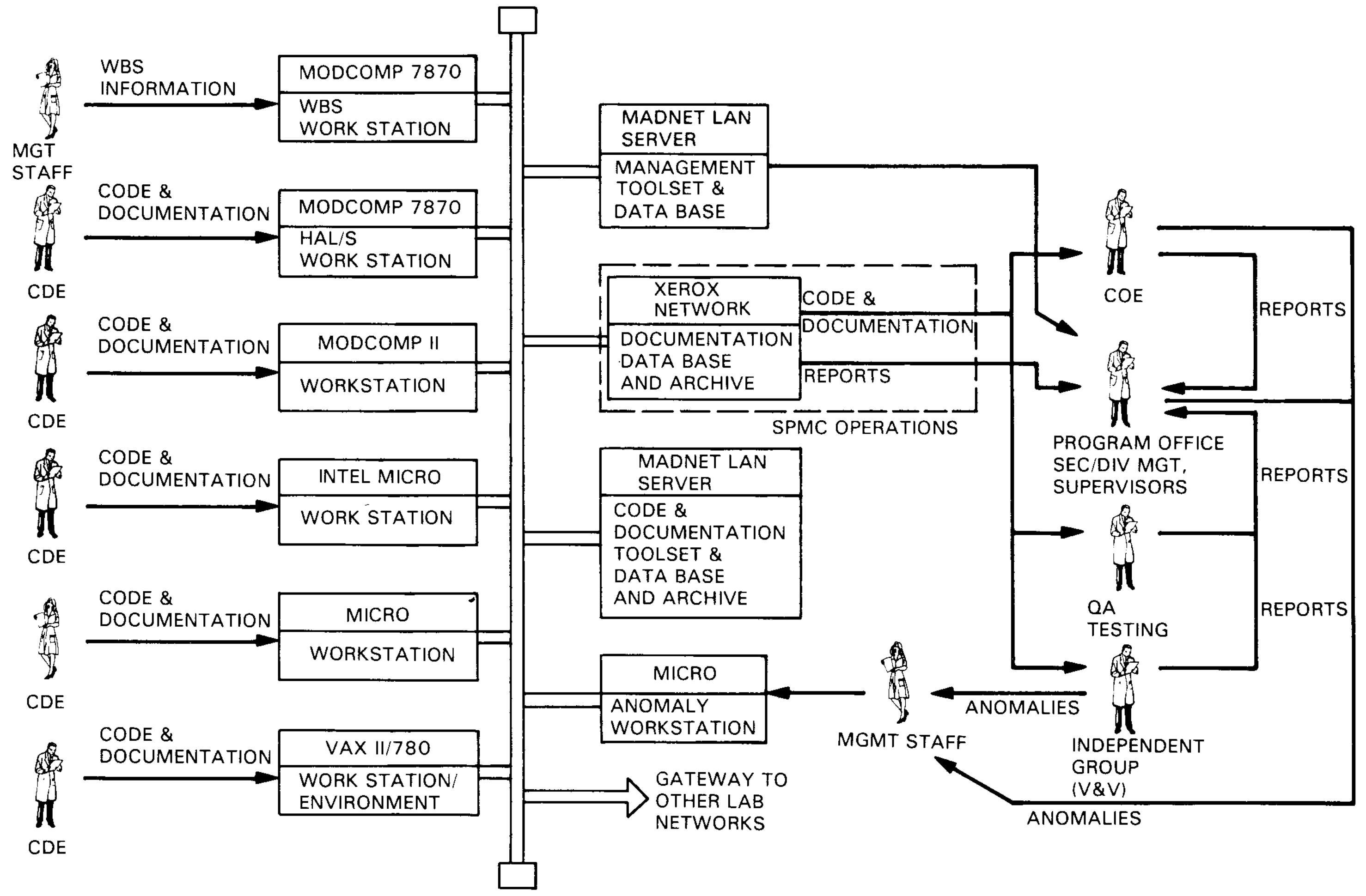

Fig. 4. Conceptual MADNET implementation process flow (from footnote 4)

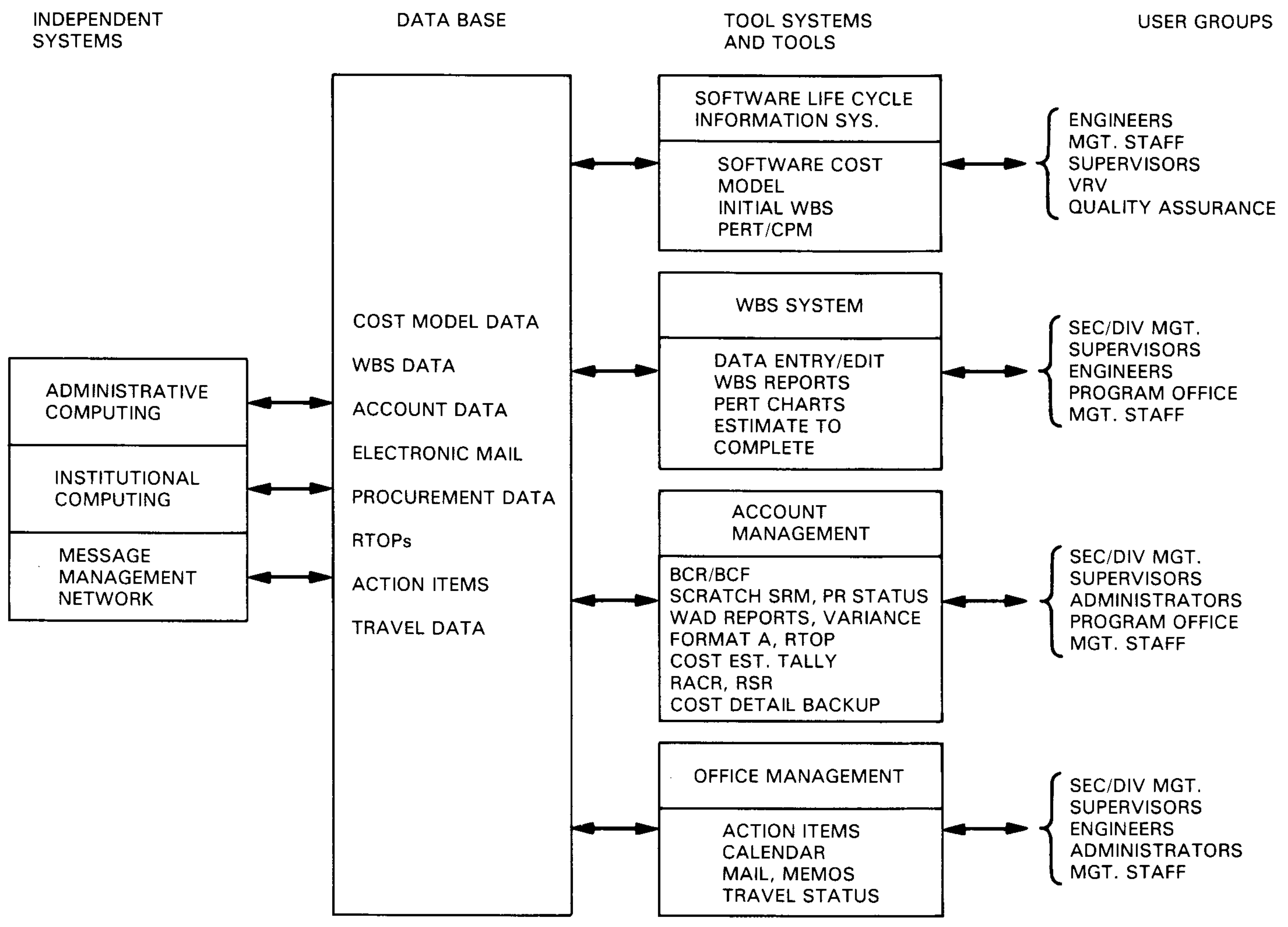

Fig. 5. Implementation data base (management components) (from footnote 4)

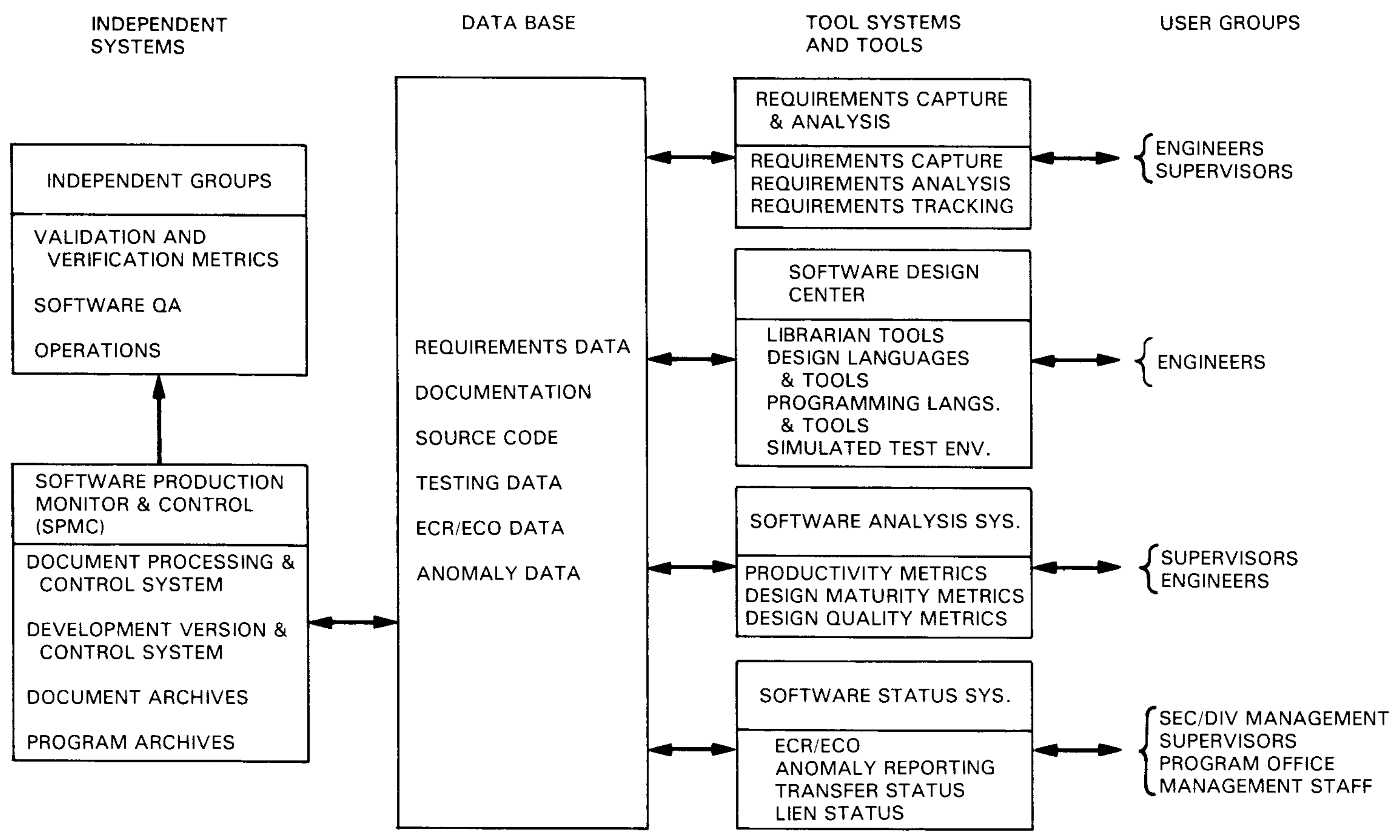

Fig. 6. Implementation data base (engineering components) (from footnote 4)

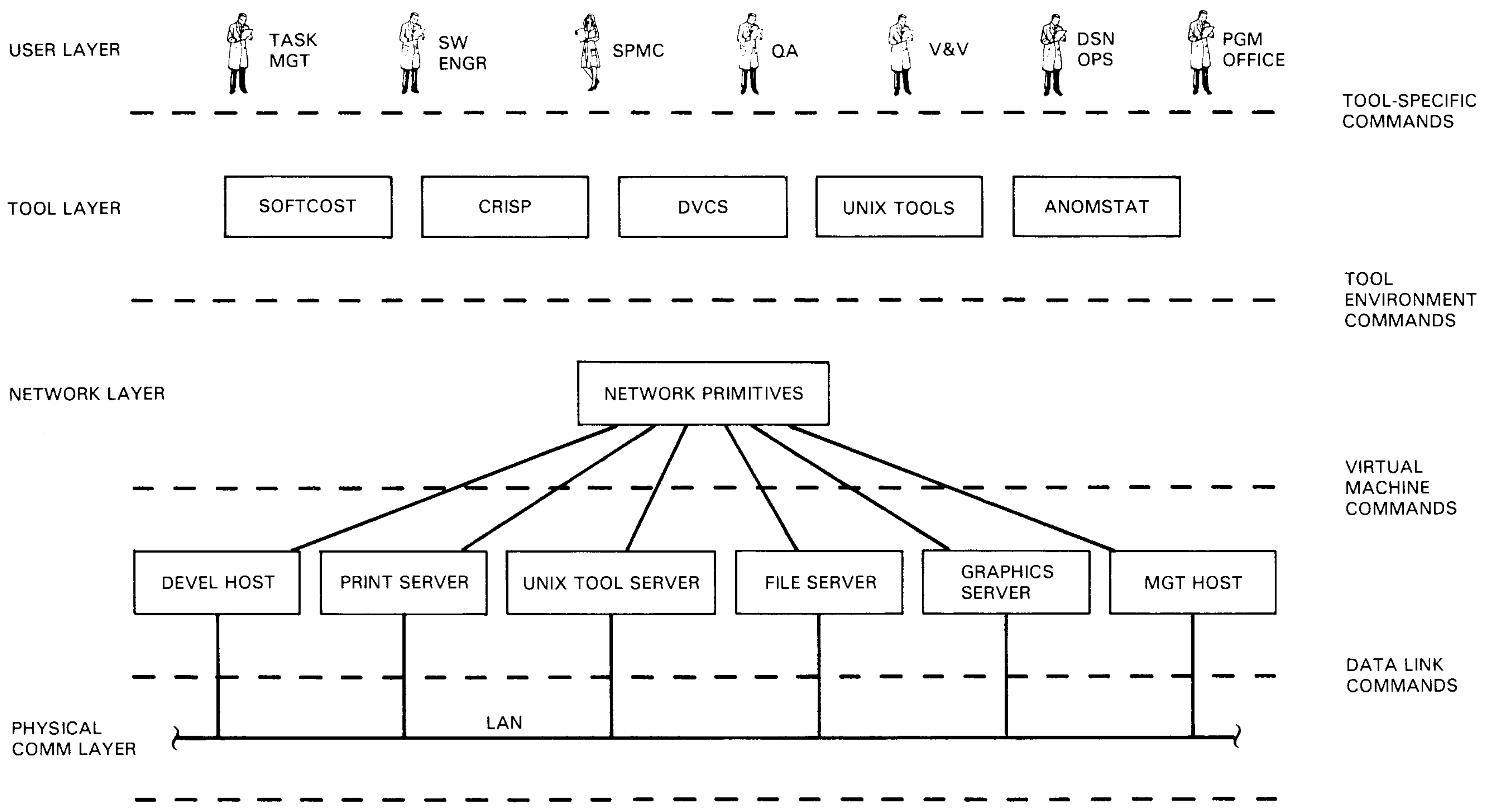

Fig. 7. Command layering via network protocols

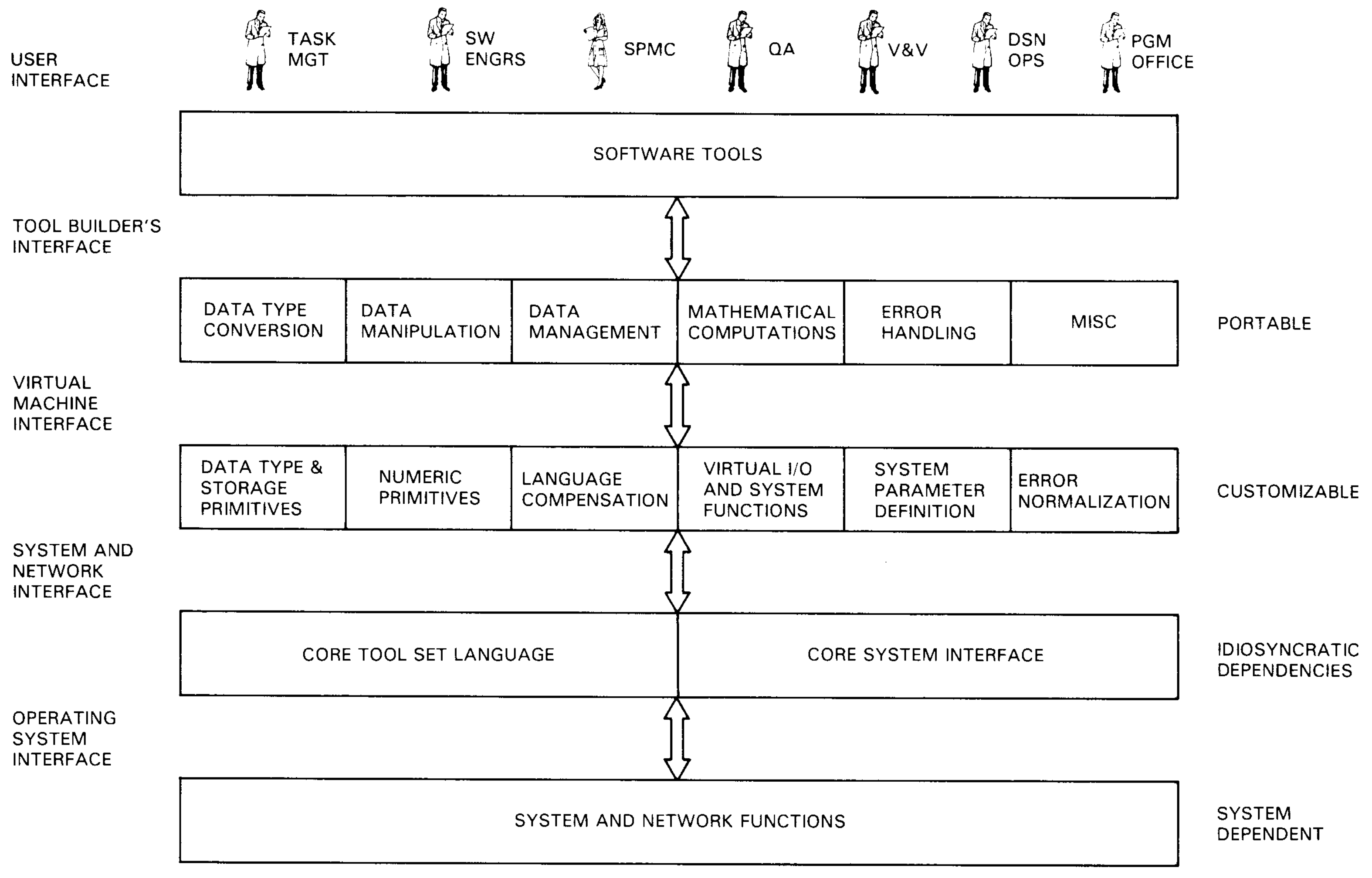

Fig. 8. Layered approach for software tools

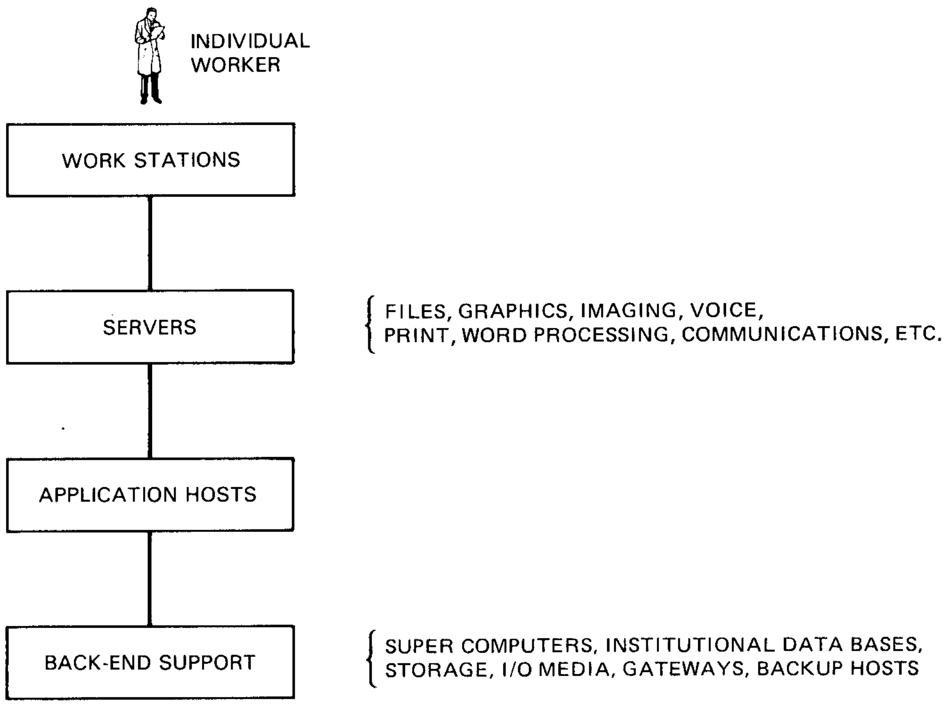
INDIVIDUAL WORKER
WORK STATIONS
SERVERS
FILES, GRAPHICS, IMAGING, VOICE, PRINT, WORD PROCESSING, COMMUNICATIONS, ETC.
APPLICATION HOSTS
BACK-END SUPPORT
SUPER COMPUTERS, INSTITUTIONAL DATA BASES, STORAGE, I/O MEDIA, GATEWAYS, BACKUP HOSTS

Fig. 9. Integrated distributed facilities

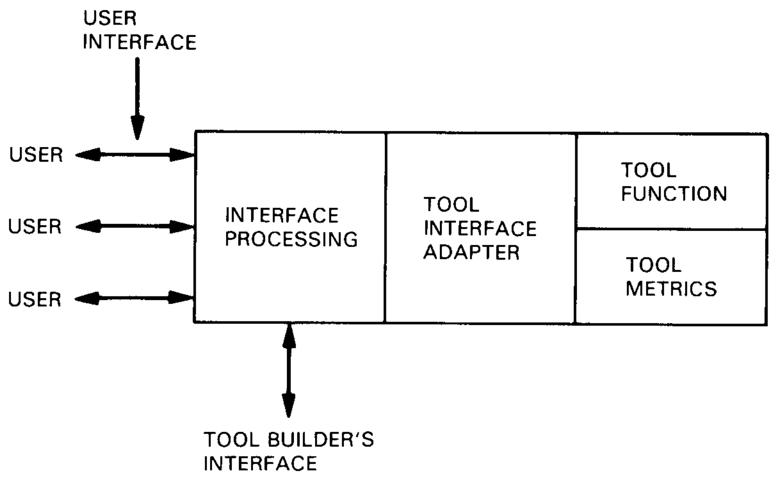

Fig. 10. Generalized tool architecture